Markov Chain Monte Carlo project

MARKOV CHAIN QUASI-MONTE CARLO

Alexandre Benharrats, Antoine Mirri

Supervised by Christian Robert

Master 2 MASH

November 2025

Contents

1 Markov Chain Monte Carlo **2**
- 1.1 Bayesian Framework . . . 2
- 1.2 MCMC principle . . . 2
  - 1.2.1 From classical Monte Carlo to Markov Chain Monte Carlo 2
  - 1.2.2 Motivations and First theorems behind MCMC principle 3
- 1.3 Metropolis-Hastings Algorithm . . . 6
  - 1.3.1 General principle . . . 7
  - 1.3.2 Formalization of the Metropolis-Hastings algorithm . . . 9
  - 1.3.3 From Uniform random samples to deterministic sequences 10

2 Quasi-Monte Carlo and Randomized Quasi-Monte Carlo **11**
- 2.1 Quasi-Monte Carlo . . . 11
  - 2.1.1 Motivations, theoretical results and limits . . . 11
  - 2.1.2 Examples of sequences with low discrepancy . . . 13
- 2.2 Randomized Quasi Monte Carlo . . . 14
  - 2.2.1 Motivations and methodology . . . 14
  - 2.2.2 Examples of randomization . . . 16
  - 2.2.3 RQMC vs Classic Monte Carlo: toy example . . . 16

3 Quasi-Monte Carlo Methods for Markov Chain Monte Carlo Sampling **18**
- 3.1 The Principle of Q-MCMC . . . 18
  - 3.1.1 Challenges in Generating CUD Sequences in Practice . . 21
  - 3.1.2 Application to Metropolis-Hastings Algorithm . . . 22
  - 3.1.3 Randomization of the QMC Metropolis-Hastings Algorithm 23
- 3.2 Implementation . . . 24
- 3.3 Conclusion . . . 26

A Annexe A : Python code for generating a CUD sequence by Tobias Schwedes **28**

B Annexe B : Python code for comparing plots of QMC sequence and IID sequence **29**

C Annexe C : Python code for comparing RQMC and MC method on toy example **31**

D Annexe D : Python code for comparing MCMC and Q-MCMC on a toy example **33**

Introduction

Monte Carlo methods address the problem of estimating integrals of the form

$$\mathbb{E}[f(X)] = \int f(x)\pi(x)\,dx,$$

where $\pi(x)$ is a target distribution and $f(x)$ is a function of interest. When analytical computation of the integral is infeasible, Monte Carlo approximates it using random samples:

$$\mathbb{E}[f(X)] \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i),$$

where $\{X_i\}_{i=1}^{N}$ are independent samples from $\pi(x)$.

Markov Chain Monte Carlo (MCMC) extends this approach to cases where direct sampling from $\pi(x)$ is impractical. By simulating a Markov chain with $\pi(x)$ as its stationary distribution, MCMC generates dependent samples that can still be used to estimate expectations. Traditionally, the transitions in MCMC are driven by streams of independent $U(0,1)$ random numbers.

In contrast, Quasi-Monte Carlo (QMC) methods offer an alternative to standard Monte Carlo simulations by employing deterministic sequences, instead of random samples. These sequences are constructed to achieve superior uniformity over the interval $[0,1]$, leading to faster convergence rates and reduced variance. Its principles can be extended to more complex stochastic methods, such as MCMC. A natural question arises: What happens when the independent random numbers used in MCMC are replaced by deterministic sequences ?

In this work, we investigate the potential of combining Quasi-Monte Carlo methods with Markov Chain Monte Carlo simulations to enhance their efficiency and performance. We begin by revisiting the classical Monte Carlo and MCMC frameworks, followed by an in-depth examination of Quasi-Monte Carlo and Randomized Quasi-Monte Carlo techniques, emphasizing their theoretical foundations and convergence properties. Finally, we introduce Quasi-MCMC with its theoretical advantages over traditional MCMC, and provide an implementation showcasing its performance on a particular example.

1 Markov Chain Monte Carlo

The principle behind MCMC algorithms is the simulation of time stationary equilibria of processes. More precisely, the Markov chain Monte Carlo algorithms deal with the problem of sampling on a given finite but huge state space and a given (stationary) distribution in an efficient way. MCMC is based on an appropriate Markov chain which converges to its stationary limit distribution. The idea of this part is to explore the theory of MCMC methods to better understand how we can adapt it to Quasi-Monte Carlo methods.

1.1 Bayesian Framework

Most applications of MCMC are oriented towards Bayesian inference. Let D denote the observed data, and θ denote model parameters and missing data. Formal inference then requires setting up a joint probability distribution $\mathbb{P}(D, \theta)$ over all random quantities. This joint distribution comprises two parts: a prior distribution $\mathbb{P}(\theta)$, which represent an inital hypothesis on the parameters before observing the data, and a likelihood $\mathbb{P}(D|\theta)$. Specifying $\mathbb{P}(\theta)$ and $\mathbb{P}(D, \theta)$ gives a full probability model, in which :

$$\mathbb{P}(D, \theta) = \mathbb{P}(D|\theta)\mathbb{P}(\theta)$$

Having observed D, Bayes theorem is used to determine the distribution of a conditional on D :

$$\mathbb{P}(D, \theta) = \frac{\mathbb{P}(\theta)\mathbb{P}(D|\theta)}{\int \mathbb{P}(\theta)\mathbb{P}(D|\theta)d\theta}$$

This is called the posterior distribution of θ , and is the object of all Bayesian inference. The posterior expectation of a function $f(\theta)$ is

$$\mathbb{E}[f(\theta)|D] = \frac{\int f(\theta)\mathbb{P}(\theta)\mathbb{P}(D|\theta)d\theta}{\int \mathbb{P}(\theta)\mathbb{P}(D|\theta)d\theta}$$

In most applications, analytic evaluation of $\mathbb{E}[f(\theta)|D]$ is impossible and requests alternative approaches as Monte Carlo integration and MCMC.

1.2 MCMC principle

1.2.1 From classical Monte Carlo to Markov Chain Monte Carlo

Monte Carlo integration evaluates $\mathbb{E}[f(X)]$ by drawing samples$(X_t)_{t=1,..,n}$ from $\pi(.)$ and then approximating with a mean $\frac{1}{n}\sum_{t=1}^{n} f(X_t)$. When the samples $(X_t)_{t=1,..,n}$ are independent, laws of large numbers ensure that the approximation can be made as accurate as desired by increasing the sample size n :

$$\frac{1}{n}\sum_{t=1}^{n} f(X_t) \xrightarrow{n\to+\infty} \mathbb{E}[f(X)]a.s.$$

and the Central Limit theorem gives us an error rate of approximation of $O(\frac{1}{\sqrt{n}})$.

However, the distribution can sometimes be quite complex to simulate, as for multivariate laws, and drawing samples X independently from $\pi(.)$ with direct methods as inversion for example is not possible.

Nevertheless, the independence of $(X_t)_{t=1,..,n}$ is not a strict requirement as long as the sampling procedure is capable of adequately exploring the support of the distribution, i.e., the region of the space where $\pi(.)$ is nonzero. To correctly estimate a property of $\pi(.)$, the frequency of the generated samples must reflect the probabilities defined by $\pi(.)$. In other words, regions where $\pi(.)$ is high should be sampled more frequently than those where $\pi(.)$ is low.

Markov Chain methods, under certain conditions, allow us to simulate correlated samples that converge after a certain time to the correct target distribution $\pi(.)$, provided that the samples are drawn across the support of $\pi(.)$ in the appropriate proportions.

1.2.2 Motivations and First theorems behind MCMC principle

The purpose of this section is to introduce the key mathematical concepts of Markov Chains that led to the development of MCMC methods, particularly the Metropolis-Hastings algorithm. Our presentation is based on the works of *Gilks* [7] and *Robert* [3], specifically Chapter 6, "Markov Chains," and Section 6.1, "Essentials for MCMC."

Markov Chain Monte Carlo is a method designed to generate samples from complex target distributions $\pi(.)$ when direct sampling is computationally infeasible. The main idea is that if a Markov chain is constructed with $\pi(.)$ as its stationary distribution, then the successive samples generated by this chain will eventually follow the distribution $\pi(.)$, even if they are not independent. To understand the foundation and motivation behind this method, we rely on a sequence of definitions and theorems that elucidate the behavior of Markov chains and their convergence properties. It is important to note that we restrict our mathematical framework to the study of **finite state-space Markov chains**. As *Hastings* (1970) observed, the practical implementation of numbers in a computer limits the Markov chains associated with Markov chain Monte Carlo (MCMC) methods to finite state spaces. Furthermore, since our primary objective is to understand the mathematical motivations behind MCMC methods, delving into the continuous case is not essential for our purposes. However, it is worth mentioning that a more in-depth exploration of these concepts necessitates an analysis of arbitrary state-space Markov chains (see [3]).

Consider a sequence of random variables $\{X_0, X_1, X_2, \dots\}$, where at each time $t \geq 0$ (discrete), the next state X_{t+1} is sampled from a conditional distribution $P(X_{t+1}|X_t)$ that depends only on the current state X_t. In other

words, given X_t, the next state X_{t+1} is independent of the history of the chain $\{X_0, X_1, \ldots, X_{t-1}\}$.

This sequence is called a *Markov chain*. Furthermore, we assume that the Markov chain is *time-homogeneous*, meaning that the transition kernel $P(X_{t+1}|X_t)$ does not depend explicitly on t.

What interests us is how to ensure the convergence in distribution of the variables, and how the initial state X_0 influences the distribution $P(X_t|X_0)$. Thus, we focus on a special type of Markov chain, called ergodic.

Definition 1.1: Ergodicity

A Markov chain with state space S, transition matrix $P = (P(X_j|X_i))_{ij\in S} = (P_{ij})_{ij\in S}$, and corresponding n-step transition matrix $P^{(n)}$ is called *ergodic* if:

1. The limits $\pi_j = \lim_{n\to\infty} P_{ij}^{(n)}$ exist for all $j \in S$,
2. These limits are positive and independent of the initial state $i \in S$,
3. The limits form a probability distribution $\pi = (\pi_1, \ldots, \pi_l)^T$, such that $\sum_{j\in S} \pi_j = 1$.

These conditions ensure that the chain's distribution will converge to a distribution $\pi(.)$. Now, the question arises: how can we characterize such Markov chains and what can guarantees that our chain is ergodic ?

Theorem 1.1

A Markov chain is *ergodic* if and only if it is *irreducible*, *aperiodic* and *positive recurrent*.

Indeed, informally speaking: A Markov chain is said to be *irreducible* if, no matter what state you start in, the kernel allows to reach any other state in the system. In other words, every state is connected to every other state in the chain, either directly or through a series of transitions. In the case of general state-space markov chain, this property also ensures that most of the chains involved in MCMC algorithms are recurrent or even Harris recurrent, which is an impoprtant notion to guarantee convergence from every starting point (see [3].

A state is called *aperiodic* if it doesn't get stuck in repeating cycles. Specifically, a state has a period if it can only be revisited at certain fixed time steps (e.g., every 2 steps or every 3 steps). If the state can be revisited at any time, then it's aperiodic. A Markov chain is *aperiodic* if all its states are aperiodic, meaning there's no fixed pattern in how the states are visited.

It is important to note that, as we work with a finite state-space the third condi-

tion is automatically verified as soon as the markov chain is irreducible. So here, only the two firsts conditions needs to both be verified to ensure convergence.

Thus, assuming that our chain satisfies these conditions, we can guarantee its convergence towards a distribution $\pi(.)$. Now, in the context of our problem, the next question is: what do we know about this distribution?

Definition 1.2: Stationarity

A probability vector $\pi = (\pi_1, \ldots, \pi_l)$ is called a *stationary distribution* for a Markov chain if it satisfies the equation $\pi = \pi P$, where P is the transition matrix. In other words : $\forall x \in S, \pi(x) = \sum_{y \in S} \pi(y)p(y, x)$

Intuitively, this means that if $X_n \sim \pi$, then $X_{n+1} \sim \pi$

Theorem 1.2: Existence and Uniqueness of the Stationary Distribution

For an irreducible and aperiodic Markov chain with a finite state space, there exists a unique stationary distribution π.

Theorem 1.3: Ergodic Theorem for Markov Chains

If a Markov chain is irreducible and aperiodic, then for any initial distribution $\mu(0)$, the distribution of the chain $\mu(n)$ converges to the stationary distribution π as $n \to \infty$.

This means that after sufficient steps, the chain "forgets" its initial state, and the samples reflect the stationary distribution.

In a general state-space setting, simple sufficient conditions on the marginal distributions and the transition kernel can ensure that the Markov chain is positive recurrent. Without these conditions, the chain risks being either null recurrent or transient, both of which prevent convergence.

Intuition Behind MCMC. Finally, we arrive at the core idea behind MCMC: suppose we can construct an irreducible and aperiodic Markov chain $\{X_n\}$ for a unique stationary distribution π. The ergodic theorem ensures that, from an arbitrary initial distribution, the chain's distribution will converge to the stationary distribution π as $n \to \infty$.

Under certain conditions (see [3]), the chain gradually "forgets" its initial state, and the distribution $P(\cdot|X_0)$ converges to the stationary distribution $\pi(\cdot)$, which no longer depends on t or X_0. Over time, the samples $\{X_t\}$ will increasingly resemble dependent samples drawn from $\pi(\cdot)$.

After a sufficient burn-in period of m iterations, the points $\{X_t; t = m + 1, \ldots, n\}$ will be approximately dependent samples from $\pi(\cdot)$. Using the output of the chain, we can now estimate the expectation $\mathbb{E}[f(X)]$, where X follows the distribution $\pi(\cdot)$. To mitigate the influence of the initial state, burn-in samples are discarded, yielding the estimator:

$$\hat{f} = \frac{1}{n-m} \sum_{t=m+1}^{n} f(X_t),$$

where m represents the number of burn-in iterations.

Thus, we understand that there are two main points to focus on:

- How to construct a chain that satisfies the conditions to have the chain distribution converging to the target distribution π.
- How to determine the necessary number of steps n to obtain a sufficient approximation, and m the number of burn in iterations.

In the following part, we will introduce one way to generate a sample of random variable following $\pi(\cdot)$ using those properties of Markov Chain.

1.3 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm was first introduced by Metropolis et al. for symmetric proposal distributions and later generalized by Hastings to allow for arbitrary proposal distributions. This algorithm is one of several methods to construct a Markov chain that converges to a given target distribution $\pi(\cdot)$.

The key idea is based on the following definition and proposition:

Definition 1.3

Let $\{X_t\}$ be a Markov chain on a finite state space S with a transition matrix P. A probability distribution $\pi(\cdot)$ on S is said to be *reversible* if, for all $x, y \in S$ with $x \neq y$,

$$\pi(x)P(x, y) = \pi(y)P(y, x).$$

Proposition

If the probability distribution $\pi(\cdot)$ is reversible for a Markov chain, then $\pi(\cdot)$ is also a stationary distribution for the chain.

Proof. For any $x \in S$, we have:

$$\pi(x) = \pi(x) \sum_{y \in S} P(x, y) = \sum_{y \in S} \pi(x)P(x, y) = \sum_{y \in S} \pi(y)P(y, x).$$

Thus, $\pi(\cdot)$ satisfies the condition of stationarity. $\square$

This proposition establishes that if we construct a Markov chain imposing the transition probabilities to satisfy the reversibility condition, the chain will have $\pi(\cdot)$ as its stationary distribution. The Metropolis-Hastings algorithm leverages this property to generate a Markov chain whose stationary distribution is the target distribution $\pi(\cdot)$.

1.3.1 General principle

Metropolis-Hastings combines two steps to generate transitions in the Markov chain.

First, starting from a state X_t, it suggests a candidate state Y using a proposal distribution $q(x, y)$. Then, it decides whether Y is accepted as the new state X_{t+1}, or if the state remains the same, i.e., $X_{t+1} = X_t$, depending on the acceptance-rejection probability $\alpha(x, y)$, which we define as:

$$\alpha(x, y) := \min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right).$$

Choice of the Acceptance Probability

We understand by construction that the total transition probability from x to y is given by:

$$P(x, y) = q(x, y)\alpha(x, y).$$

We can choose $\alpha(x, y)$ as:

$$\alpha(x, y) = \min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right).$$

The transition matrix defined by the Metropolis-Hastings algorithm with this choice of α specifies a reversible Markov chain with stationary distribution $\pi(\cdot)$.

Proof. To prove this, we need to establish that the detailed balance condition holds, which states:

$$\pi(x)P(x, y) = \pi(y)P(y, x), \quad \forall x, y \in S.$$

This condition ensures that the chain is reversible and that $\pi(\cdot)$ is a stationary distribution. Let us consider the case where $x \neq y$, as the case $x = y$ follows trivially from the definition of the transition probabilities.

From the construction of the Metropolis-Hastings algorithm, the transition probability $P(x, y)$ is given by:

$$P(x, y) = q(x, y)\alpha(x, y),$$

where $q(x, y)$ is the proposal distribution and $\alpha(x, y)$ is the acceptance probability, defined as:

$$\alpha(x, y) = \min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right).$$

Substituting the transition probability into $\pi(x)P(x, y)$, we have:

$$\pi(x)P(x, y) = \pi(x)q(x, y)\alpha(x, y) = \pi(x)q(x, y)\min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right)$$

The minimum function $\min(1, a)$ ensures that either the first term or the second term in the comparison determines the value, leading to:

$$\pi(x)P(x, y) = \min\left(\pi(x)q(x, y), \pi(y)q(y, x)\right).$$

Similarly, consider the reverse transition $\pi(y)P(y, x)$:

$$\pi(y)P(y, x) = \pi(y)q(y, x)\min\left(1, \frac{\pi(x)q(x, y)}{\pi(y)q(y, x)}\right) = \min\left(\pi(y)q(y, x), \pi(x)q(x, y)\right).$$

In conclusion, $\pi(x)P(x, y) = \pi(y)P(y, x)$, as the minimum function is symmetric in its arguments. This satisfies the detailed balance condition. So the Markov chain defined by the transition matrix is reversible. Furthermore, the reversibility implies that $\pi(\cdot)$ is a stationary distribution of the Markov chain. □

Indeed, choosing this alpha impose the reversibility condition $\pi(x)P(x, y) = \pi(y)P(y, x)$ on our target probability distribution, and we have the following result (see [7]) :

$$\pi(X_t)P(X_t, X_{t+1}) = \pi(X_{t+1})P(X_{t+1}, X_t)$$

Integrating both sides with respect to X_t yields:

$$\int \pi(X_t)P(X_t, X_{t+1})\, dX_t = \pi(X_{t+1}).$$

The left-hand side of this equation represents the marginal distribution of X_{t+1} under the assumption that X_t is drawn from the stationary distribution $\pi(\cdot)$. Thus, this equation confirms that if the current state X_t is distributed according to $\pi(\cdot)$, then the next state X_{t+1} will also follow the same distribution $\pi(\cdot)$.

In total, we have that once a sample is drawn from the stationary distribution $\pi(\cdot)$, all subsequent samples obtained through the Markov chain transitions will also follow $\pi(\cdot)$, provided the chain is correctly constructed and satisfies reversibility and irreducibility. However this proof establishes that $\pi(\cdot)$ is indeed a stationary distribution for the Markov chain but does not prove that the chain converges to $\pi(\cdot)$ from an arbitrary initial distribution. Convergence requires additional properties as we mention before.

1.3.2 Formalization of the Metropolis-Hastings algorithm

Metropolis-Hastings algorithm

1. Choose an arbitrary initial point X_0 and a proposal distribution $q(x, y)$ that generates candidate points Y given the current state X_t. 2. At each time step t:

   - Sample a candidate point Y from the proposal distribution $q(\cdot|X_t)$.
   - Compute the acceptance probability:

$$\alpha(X_t, Y) = \min\left(1, \frac{\pi(Y)q(Y, X_t)}{\pi(X_t)q(X_t, Y)}\right).$$

   - Sample a uniform random variable $U \sim \text{Uniform}(0, 1)$.
   - If $U \leq \alpha(X_t, Y)$, accept the candidate point and set $X_{t+1} = Y$. Otherwise, reject the candidate point and set $X_{t+1} = X_t$.

3. Increment t and repeat.

Remarks

- More informally, the intuition behind this choice of alpha is that,
  - If $\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)} \geq 1$, then $\alpha(x, y) = 1$. This ensures that transitions to states with higher target density $\pi(\cdot)$ are always accepted.
  - If $\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)} < 1$, the candidate is accepted with probability proportional to the relative densities. This maintains a balance and ensures the chain explores the state space properly.
- In practice, the candidate point Y is often generated from a uniform distribution on $[0, 1]^d$. This can be achieved using the inverse transform sampling method or other appropriate sampling techniques depending on the chosen proposal distribution.
- The support of the proposal distribution $q(x, y)$ must include the support of the target distribution $\pi(x)$. If $q(x, y)$ does not cover the entire support of $\pi(x)$, certain states will never be reached, preventing convergence to $\pi(\cdot)$.
- The target distribution $\pi(x)$ needs to be known only up to a normalizing constant. For example, if $\pi(x) = \frac{g(x)}{\int g(t)dt}$, the ratio $\frac{\pi(y)}{\pi(x)}$ simplifies to $\frac{g(y)}{g(x)}$, making the algorithm feasible without computing the integral.

- The chain can revisit the same state multiple times, meaning the samples are not independent and identically distributed (non-i.i.d.). However we can choose the proposal distribution q such that the candidate Y doesn't depend on X_t. It is called the independence sampler.
- Initial samples (burn-in) are discarded to mitigate the influence of the arbitrary starting point X_0 and allow the chain to reach its stationary distribution.

1.3.3 From Uniform random samples to deterministic sequences

The Metropolis-Hastings algorithm, as described above, relies on sampling independent and identically distributed (i.i.d.) uniform random variables $U \sim \text{Uniform}(0, 1)$, either to build a random variable proposal candidate Y and to decide whether to accept or reject the proposed candidate points. These i.i.d. random variables ensure the required randomness and independence at each iteration, allowing the Markov chain to sample from the target distribution. However, it is possible to explore an alternative approach by replacing these i.i.d. random variables with deterministic sequences, designed to cover the interval $[0, 1]$ more uniformly, resulting in improved convergence rates and reduced variance. This method is called *Quasi-Monte Carlo*.

2 Quasi-Monte Carlo and Randomized Quasi-Monte Carlo

2.1 Quasi-Monte Carlo

2.1.1 Motivations, theoretical results and limits

We remind that our goal is to estimate the quantity $\mu = \int_{[0,1]^d} f(\mathbf{u})\,\mathrm{d}\mathbf{u}$.

In this section, we directly assume that f contains the transformation of the uniform random variable $\mathbf{u}$ and the measurable function in the expectation. A way of estimating this quantity is by using a classical MC method and using the following estimate: $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{u}_i)$ where the $\mathbf{u}_1, ..., \mathbf{u}_n$ are pseudo-random generations of law $\mathcal{U}[0,1]^d$. As we previously said in 1.2.1, we obtain an unbiased estimator, with an almost surely convergence (LLN), confidence intervals and the RMSE is $O(n^{-1/2})$ (CLT).

The idea of QMC is still to estimate our target with $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{u}_i)$, but now the $\mathbf{u}_1, ..., \mathbf{u}_n$ are deterministic points carefully chosen to be "more uniformly distributed" than the previous random points.

We quantify the "degree of uniformity" of a sequence $\mathbf{x}_1, ..., \mathbf{x}_n \in [0,1]^d$ with the notion of **discrepancy**. The most relevant one is the following :

Definition 2.1

Star discrepancy of $\mathbf{x}_1, ..., \mathbf{x}_n \in [0,1]^d$

$$D_n^* = D_n^*(x_1, \ldots, x_n) \sup_{a \in [0,1]^d} |\delta(a; x_1, \ldots, x_n)| .$$

Local discrepancy of $\mathbf{x}_1, ..., \mathbf{x}_n$ at $\mathbf{a} \in [0,1]^d$:

$$\delta(\mathbf{a}) = \delta(\mathbf{a}; x_1, \ldots, x_n) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{x_i \in [0,\mathbf{a})^d} - \prod_{j=1}^{d} a_j.$$

Here, the local discrepancy would be small if the fraction of our n points in $[0,a)^d$ match the "volume" of $[0,a]^d$ which is between 0 and 1. Then, we understand that when D_n^* is small, the fraction of our n points in each "box" is close to the proportion of the unit cube taken up by that box. So we want to create $\mathbf{x}_1, ..., \mathbf{x}_n$ with low star discrepancy. We characterize this statement with the following definition :

Definition 2.2

The infinite sequence $x_1, x_2, \cdots \in [0,1]^d$ is **uniformly distributed** if

$$D_n^*(x_1, \ldots, x_n) \to 0 \quad \text{as } n \to \infty.$$

As we no longer work with random variables but with deterministic sequences, we can't use the Law of Large Number and the Central Limit theorem to assure convergence and estimate the error. Following [5], we obtain the two following theorems which are quite equivalent for the deterministic QMC estimates:

Theorem 2.1

Let f be a Riemann integrable function on $[0,1]^d$. If $x_1, x_2, \cdots \in [0,1]^d$ are uniformly distributed, then

$$\left| \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}_i) - \int_{[0,1]^d} f(\mathbf{x})\, \mathrm{d}\mathbf{x} \right| \to 0 \quad \text{as } n \to \infty.$$

Theorem 2.2 : Koksma-Hlawka inequality

For $d \geq 1$ and $x_1, \ldots, x_n \in [0,1]^d$,

$$\left| \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}_i) - \int_{[0,1]^d} f(\mathbf{x})\, \mathrm{d}\mathbf{x} \right| \leq D_n^*(x_1, \ldots, x_n) V_{\mathrm{HK}}(f),$$

where $V_{\mathrm{HK}}(f)$ denotes the total variation of f in the sense of Hardy and Krause.

The problem of the last theorem is that it uses the unknown quantities D_n^* and $V_{\mathrm{HK}}(f)$. But it is still an extremely important result as we introduce an additional notion over the discrepancy of a sequence:

Definition 2.3

The infinite sequence $x_1, x_2, \cdots \in [0,1]^d$ is a **low discrepancy sequence** if

$$D_n^*(x_1, \ldots, x_n) = O(n^{-1} (\log n)^d) \quad \text{as } n \to \infty.$$

Following [5], it implies that asymptotically, a low discrepancy sequence has $D_n^*(x_1, \ldots, x_n) = O(n^{-1+\epsilon})$ for any $\epsilon \geq 0$ as $n \to \infty$. Thus if we create a low discrepancy sequence and have $V_{\mathrm{HK}}(f) < \infty$ we obtain $|\hat{\mu} - \mu| = O(n^{-1+\epsilon})$. Finally we get the result that if n is large enough a QMC method using a low-

discrepency sequence should achieve better accuracy compared to $O(n^{-1/2})$ of classical Monte Carlo.

It is important not to forget the assumption $V_{\mathrm{HK}}(f) < \infty$ and the fact that this holds for extremely large n when we get to high dimensions (in reality we have $O(n^{-1}(\log n)^d)$ which implies difficulties for large d). In addition, the estimate is biased and we can't compute error estimation because QMC is deterministic.

2.1.2 Examples of sequences with low discrepancy

In this section, we introduce a few examples of low-discrepancy sequences and discuss the limitations they encounter when applied to MCMC. We aim to provide a general overview and we will not delve into the full range of construction methods or their theoretical foundations, as this constitutes a different field of research.

The Van der Corput sequence

For an integer $b \geq 2$, the sequence is defined by $x_i = \phi_b(i-1)$ for $i \geq 1$, where:

$$\phi_b(i) = \sum_{k=0}^{\infty} d_{k,b}(i) b^{-k-1}$$

and $d_{k,b}(i) \in \{0, 1, \ldots, b-1\}$ is the coefficient of b^k in the base b expansion of i.

The idea of the ϕ function is to map the nonnegative integers in $[0, 1)$. The Van der Corput sequences have low discrepancy but they have limitations. First, they only live in d=1 and we need QMC methods for high dimensions. This sequence is also a good example that we need more than low discrepancy conditions if we want to use QMC sequences in MCMC.

Indeed, MCMC is by construction sequential and some QMC sequences are not. One can see that for $(u_n)_n$ a Van der Corput sequence we have $u_{2n} \in (0, 1/2)$ and $u_{2n-1} \in [1/2, 1)$ $\forall n$. So if we want to use this sequence in d=2 we could create the sequence of overlapping tupels $u_i^{(2)} = (u_i, u_{i+1})$ but then:

$$\mathbf{u}_{2n}^{(2)} \in (0, 1/2) \times [1/2, 1), \quad \text{and} \quad \mathbf{u}_{2n-1}^{(2)} \in [1/2, 1) \times (0, 1/2).$$

This sequence never hits the square $(0, 1/2)$ x $(0, 1/2)$ thus is not of low discrepancy and the combination with MCMC fails (see Caflisch and Moskowitz (1995) [4]). We will later introduce sequences with stronger properties that suit the combination with MCMC.

We now present a sequence that generates points in dimension d with low discrepancy.

The Halton sequence

The sequence generates points $\mathbf{x}_1, ..., \mathbf{x}_n \in [0,1)^d$ as follows:

$x_{ij} = \phi_{p_j}(i-1), \quad i \geq 1, \quad 1 \leq j \leq d,$ where p_j is the j'th prime number

We compare bellow in d=2 pseudo-random generations and the Halton sequence (see *Appendix B*):

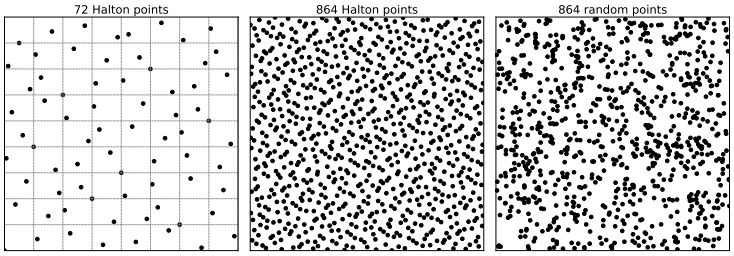

Figure 1: Halton sequence in the unit square: Left panel shows 72 Halton points, middle panel shows 864 Halton points, and right panel shows 864 random points.

There are many other similar sequences with sometimes better discrepancy bounds (which is important for Theorem 2.2). Considering that Halton sequences use more prime numbers as dimensionality increases, the theory says that for d=5 the Halton sequence needs $2*3*5*7*11 = 2310$ points to obtain acceptable stratification (for d = 10, we need 6,469,693,230 points as the product of the first ten primes).

Other methods for constructing sequences with low discrepancy exist such as digital nets which work better with high dimension problems. We also mention other methods called *lattices* that are efficient for integrating periodic functions and are faster than MC for these kinds of problem. We will not cover these methods as this part is only an hint of how QMC sequences are constructed (see [5]).

2.2 Randomized Quasi Monte Carlo

2.2.1 Motivations and methodology

Randomized Quasi-Monte Carlo (RQMC) combines the strengths of Quasi-Monte Carlo (QMC) and Monte Carlo (MC). In RQMC, the points $x_1, \ldots, x_n$ are individually distributed as $U[0,1]^d$, while collectively exhibiting low discrepancy with probability one. This approach allows for practical error quantification via confidence intervals obtained from independent replications.

Following [2], RQMC methods reduce the root mean square error (RMSE) to $O(n^{-3/2+\epsilon})$ in some cases, significantly better than the standard MC rate of $O(n^{-1/2})$. Unlike QMC, which achieves an error rate of $O(n^{-1+\epsilon})$ under strong regularity assumptions, RQMC introduces randomness to address challenges like computing the total variation of f or the star discrepancy of QMC points.

The idea of RQMC is to apply a randomization to a low discrepancy QMC sequence $a_1, \ldots, a_n$. We get the following definition:

Definition 2.4

Random variables $x_i \in [0,1]^d$ comprise a **randomized quasi-Monte Carlo** rule if there exist $B < \infty$ and $N > 0$ s.t:

$$\mathbb{P}\left(D_n^*(x_1, \ldots, x_n) < B\frac{(\log n)^d}{n}\right) = 1 \quad \text{for all } n \geq N$$

and $x_i \sim \mathcal{U}[0,1]^d, \quad \text{for all } i \geq 1.$

Points $x_1, \ldots, x_n$ in a RQMC rule are individually uniformly distributed on $[0,1]^d$ and collectively have a low star discrepancy. We then get the usual average estimate of $\mu = \int_{[0,1]^d} f(\mathbf{x})\,\mathrm{d}\mathbf{x}$ with:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{x}_i)$$

We have an unbiased estimate for which we can compute the variance when f has bounded variation:

$$\mathrm{Var}(\hat{\mu}) = \mathbb{E}((\hat{\mu} - \mu)^2) \leq \mathbb{E}((D_n^*(x_1, \ldots, x_n)V_{\mathrm{HK}}(f))^2) < B^2 V_{\mathrm{HK}}(f)^2 \frac{\log(n)^{2d}}{n^2}.$$

Thus for large enough n, RQMC is asymptotically better than MC but we still have the variation of f in the bound. The randomization of $a_1, \ldots, a_n$ into $x_1, \ldots, x_n$ can be performed independently R times and we finally get the pooled estimate with its variance estimate (requiring nR function evaluation):

$$\hat{\mu}_{\mathrm{pool}} = \frac{1}{R}\sum_{r=1}^{R} \hat{\mu}_r$$

$$\widehat{\mathrm{Var}}(\hat{\mu}_{\mathrm{pool}}) = \frac{1}{R(R-1)}\sum_{r=1}^{R}(\hat{\mu}_r - \hat{\mu}_{\mathrm{pool}})^2.$$

Methods for choosing R and decision between two different RQMC methods are developed in [5].

2.2.2 Examples of randomization

Let $a_1, \dots, a_n \in [0,1]^d$ a sequence with low discrepancy, we introduce a simple form of randomization, the **Cranley-Patterson rotation**:

$$x_i = a_i + u - \lfloor a_i + u \rfloor$$

Where $\mathbf{u} \sim \mathcal{U}[0,1]^d$ is the rotation vector. The resulting $x_1, \dots, x_n$ form a RMQC rule and the results presented before hold. This randomization is commonly applied to lattice rules. The resulting bounds and consistency of such randomization are developed in details in [5].

The main limit of the Cranley-Patterson rotation applied to a sequence with low discrepancy is that it does not preserve the properties that define this sequence most of the time. A classic example is with digital-nets (mentionned in 2.1.2). To preserve stratification properties of such sequences, other randomization methods have been developed such as strategic randomization of the digits of the points called **scrambling methods** (see [5] for more details).

2.2.3 RQMC vs Classic Monte Carlo: toy example

Now that we have recover some error quantification with the randomization in RQMC, we can compare such methods with classical Monte-Carlo on a toy example.

The function we aim to integrate is:

$$f(x) = \prod_{i=1}^{d} \cos(x_i),$$

where $x \in [0,1]^d$ and d represents the dimension. This function is smooth and periodic, making it suitable for highlighting the advantages of low-discrepancy sequences used in RQMC methods.

The exact value of the integral can be computed analytically as:

$$I = (\sin(1) - \sin(0))^d,$$

We then run each method 200 times with 16384 points each time. We compare the classical Monte-Carlo with Halton and Sobol RQMC (using scramble randomization).

Table 1 summarizes the results of the comparison (see *Appendix C* for code):

Method	Dimension	Pool Estimate	MSE	Exact Value
Monte Carlo	2	7.08e-01	1.67e-06	7.08e-01
RQMC Halton	2	7.08e-01	3.10e-16	7.08e-01
RQMC Sobol	2	7.08e-01	3.18e-10	7.08e-01
Monte Carlo	24	1.59e-02	1.41e-08	1.59e-02
RQMC Halton	24	1.59e-02	2.56e-10	1.59e-02
RQMC Sobol	24	1.59e-02	4.92e-10	1.59e-02

Table 1: Comparison of Monte Carlo and RQMC methods.

We observe that for a smooth function as this one, both RQMC methods performe better than the classic MC, even in large dimension.

3 Quasi-Monte Carlo Methods for Markov Chain Monte Carlo Sampling

This section aims to explore the adaptation of Quasi-Monte Carlo (QMC) methods to samplers based on Markov Chain Monte Carlo (MCMC) techniques, particularly the Metropolis-Hastings algorithm. Unlike the relatively straightforward adaptation of QMC to simple Monte Carlo (MC) methods, integrating QMC with MCMC poses significant challenges. Recall that in a Metropolis-Hastings scheme, each step in the Markov chain requires two stochastic operations using IID random variables: the generation of a proposal value y from a transition proposal distribution, and the acceptance or rejection of that proposal based on the computed acceptance probability.

The goal is to develop QMC sequences tailored to the MCMC framework and to evaluate the performance of this approach by replacing the standard independent and identically distributed (IID) random samples with randomized QMC (RQMC) sequences. This section builds on the work of *Seth D. Tribble* and Art B. Owen in [9] and [2].

Two critical considerations highlighted in their work are as follows:

1. The Complexity of "Superiority": Determining whether QMC sequences outperform IID sampling in the context of MCMC is highly nontrivial. Unlike classical MC settings, the dependence introduced by successive updates in the Markov chain diminishes the benefits of using low-discrepancy sequences. This introduces a new form of the "curse of dimensionality," which limits the advantages that QMC sequences provide in classical MC applications.

2. Purpose of Q-MCMC: The goal of Q-MCMC is not to accelerate the convergence of the Markov chain to its stationary distribution π. Instead, it seeks to improve the accuracy of the estimates derived from the chain, analogous to the role QMC plays in standard Monte Carlo methods.

By addressing these challenges, this section provides insight into the theoretical and practical implications of integrating QMC with MCMC methods and evaluates the potential benefits of this hybrid approach.

3.1 The Principle of Q-MCMC

Consider a d-dimensional MCMC sampler where each step requires up to d independent uniform random variates to generate a proposal and compute the acceptance/rejection decision. After a burn-in period, the MCMC algorithm generates N samples, and the required univariate random values can be orga-

nized into a variate matrix:

$$\begin{bmatrix} u(1) & u(2) & \dots & u(d) \\ u(d+1) & u(d+2) & \dots & u(2d) \\ \vdots & \vdots & \ddots & \vdots \\ u((N-1)d+1) & u((N-1)d+2) & \dots & u(Nd) \end{bmatrix}.$$

The sequence $\{u(1), u(2), \dots, u(Nd)\}$ is referred to as the *driving sequence* of the MCMC algorithm. Each row of the variate matrix generates one step in the Markov chain. Specifically, the m-th sample is determined by:

$$X^{(m)} = \phi\left(X^{(m-1)}, u_{(m-1)d+1:md}\right),$$

where ϕ is the transition function depending on the current state $X^{(m-1)}$ and the d uniform variates in the corresponding row and $u_{i:j} := (u(i), u(i+1), ..., u(j))$. This formalization directly comes from [9].

As previously mentioned, it is essential for the effectiveness of sampling that points generated in blocks of d fill the unit hypercube $[0,1)^d$ uniformly. However, in the context of an MCMC sampler, where each QMC point is used to generate a single step of the Markov chain, many QMC point sets can produce inaccurate results (see [9]). This highlights that the low-discrepancy property, as defined in the previous section, is no longer sufficient under these constraints.

To address this issue, we require sequences where blocks of d successive variates fill the hypercube $[0,1)^d$ uniformly. Asymptotically, this means achieving a discrepancy that approaches zero. To formalize this, we introduce the concept of **Completely Uniformly Distributed (CUD)** sequences.

Definition 3.1 : Completely Uniformly Distributed Sequence

A sequence $\{u_1, u_2, \dots\}$ is said to be *completely uniformly distributed* (CUD) if, for every dimension $d \geq 1$, the discrepancy D_n^* of the sequence satisfies:

$$\lim_{n\to\infty} D_n^*(z_1, z_2, \dots, z_n) = 0,$$

where $z_i = (u_i, u_{i+1}, \dots, u_{i+d-1})$ are d-dimensional points.

The following lemma ensures that blocks of variates generated by the sequence achieve asymptotic uniformity, whether they are constructed in a successive or independent manner. More specifically, it guarantees that even when examining successive subsets of the sequence, these subsets will fill the unit hypercube uniformly as the sequence grows larger.

Lemma

A sequence $\{u(1), u(2), \dots\}$ is CUD if and only if, for arbitrary integers $d \geq l \geq 1$, the sequence of d-tuples $\{z(i)\}$ defined by $z(i) = (u(id - l + 1), u(id - l + 2), \dots, u(id - l + d))$ satisfies:

$$\lim_{n\to\infty} D_n^* \left(z(1), z(2), \dots, z(n)\right) = 0.$$

Our grouping uses $l = d$, where d is the block size. The more general result in the lemma allows skipping the first $d - l$ values u_i, which provides more flexibility in constructing the blocks. It highlights the robustness of the CUD property, ensuring a good balance in the distribution of points, whether they are generated in a correlated or independent manner.

The use of CUD sequences in Markov chains has been shown to ensure consistency under certain conditions. A generalization of Chentsov's theorem establishes this result for finite state spaces, assuming that all transition probabilities are strictly positive.

Theorem 3.1 Consistency with CUD Sequences

Suppose the state space is finite, $S = \{\omega_1, \dots, \omega_K\}$, and the sequence $u^{(1)}, u^{(2)}, \dots$ is used to run a Metropolis-Hastings sampler with regular homogeneous proposals. Assume the resulting sample is weakly consistent if the $u^{(i)}$ are IID $U[0, 1)$, such that the following holds:

$$\frac{1}{N}\sum_{n=1}^{N} f(X_n) \xrightarrow[N\to\infty]{} \int_S f(x)\pi(x)\,dx, \tag{1}$$

where $\pi(x)$ is the stationary distribution of the Markov chain.
(see proof [9])
Then, if the $u^{(i)}$ form a Completely Uniformly Distributed (CUD) sequence, the consistency result in Eq. 1 still holds.

While it is true that certain non-CUD sequences can still ensure consistency, it is equally straightforward to construct examples of Metropolis-Hastings samplers where a non-CUD sequence would fail to meet this requirement. Therefore, to ensure robustness and adaptability to arbitrary sampling schemes, CUD sequences are the most appropriate choice for MCQMC methods.

Note that this theorem impose a pretty strong assumption on S which must be finite. However a generalization made by S. CHEN, J. DICK and A. B. OWEN in [1] states that MCQMC algorithms formed by Metropolis–Hastings updates driven by CUD points can consistently sample a continuous stationary distribution.

3.1.1 Challenges in Generating CUD Sequences in Practice

The generation of Completely Uniformly Distributed (CUD) sequences poses significant practical challenges, particularly due to the sequential nature of traditional Random Number Generators (RNGs). The purpose of this section is not to present an optimal method for easily generating CUD sequences but rather to provide some insights and, more importantly, to highlight a challenge that does not arise in classical MCMC methods.

According to [9] and [8], most RNGs are designed to produce pseudo-random sequences with a finite period N, and their output eventually repeats. This periodicity inherently violates the CUD criterion, which requires the discrepancy of blocks to asymptotically decay to zero across the entire sequence.

To address this issue, one cannot rely solely on treating the output of a single cycle of an RNG as a valid CUD sequence. Instead, two primary strategies have been proposed to construct sequences that approximate the desired CUD properties in practical settings:

1. **Using RNGs with Growing Periods:** Rather than relying on a single RNG with a fixed period, one can use classes of RNGs with increasingly larger periods. If the sequence outputs of these generators satisfy an *array-CUD* property, then they can be justified for use in MCQMC algorithms.

2. To emulate CUD properties within the periodic constraints of traditional RNGs, we can use the **variate matrix representation** showed previously, where the rows are used to update the Markov chain, as the generator's output. By repeating the generator's output d times and ensuring that N (the period of the generator) and d (the dimension of the variate matrix) are relatively prime, the columns of this matrix include each value exactly once. This ensures a balanced distribution across the variates, maintaining a uniform coverage of the space $[0, 1)^d$ and reducing the impact of periodicity.

. While these approaches do not eliminate the inherent limitations of RNGs, they provide practical methods to achieve sequences that approximate CUD behavior and can be used in Metropolis-Hastings within the MCQMC framework. One can show that an MCQMC algorithm which replaces IID sampling with points drawn from thoose generator is valid in an array-consistency sense (see *Chapter 4* [9])

In our implementation presented in the final section, we utilized a generator of CUD sequences based on the second approach described earlier. This generator, presented in *Appendix A*, was obtained from [6] and is the same as the one used by Tobias Schwedes and Ben Calderhead in [8], which may explain the similar results we will present later.

The following plot represent a 2D exemple of the points generated by this generator.

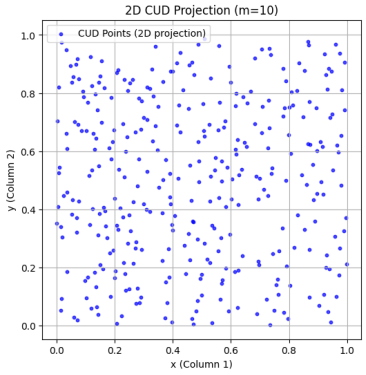

Figure 2: CUD sequence in the unit square for 1023 points generated

We observed a good uniformity over the cube space. Even though the algorithm generates 1023 points for m=10, we only have ploted 682 points, according to our matrix representation, as explained in the follwing part.

3.1.2 Application to Metropolis-Hastings Algorithm

Using CUD sequences adapted to the Metropolis-Hastings algorithm, we can rewrite the method as follow :

QMC Metropolis-Hastings Algorithm

The proposal y_{i+1} can be written as:

$$y_{i+1} = \Phi_i(X_i, u_{di+1}, \ldots, u_{di+d-1}),$$

where Φ_i is a transformation function based on the current state X_i and $d-1$ uniform random variables.

The next state is determined as:

$$X_{i+1} = \begin{cases} y_{i+1}, & \text{if } u_{di+d} \leq \alpha(X_i, y_{i+1}), \\ X_i, & \text{otherwise,} \end{cases}$$

where $\alpha(X_i, y_{i+1})$ is the acceptance probability.

Note that for a homogeneous sampler, Φ_i and A do not depend on i.

By organizing the sequence into this structure, we effectively distribute the uniform random sequence across dimensions required for the Markov chain generation. This ensures a seamless integration of the CUD sequence into the Metropolis-Hastings algorithm.

Back to our previous example on *Figure 2*, we have the following representation of our CUD sequence :

$$\begin{bmatrix} u(1) & u(2) & u(3) \\ u(4) & u(5) & u(6) \\ \vdots & \vdots & \vdots \\ u(3N-2) & u(3N-1) & u(3N) \end{bmatrix}.$$

In this matrix where each row corresponds to one step in the Markov chain generation process :

- The first two columns $u(1)$ and $u(2)$ are used to generate proposals for the Markov chain.
- The third column $u(3)$ is utilized for the acceptance-rejection step in the Metropolis-Hastings algorithm.

In *Figure 2*, we only plot the coordonates of the 2D CUD points we have generated. We didn't plot here the third column ($u(3)$ which utilized for the acceptance-rejection step in the Metropolis-Hastings algorithm.

3.1.3 Randomization of the QMC Metropolis-Hastings Algorithm

The goal of randomization in the sampling process is to improve its statistical properties by reducing bias and preserving the CUD low-discrepancy property, specifically the array-wise. In order to achieve this, randomization is applied uniformly to each row of the variate matrix. This ensures that the d-dimensional points in each row remain marginally uniformly distributed in the unit cube $[0,1)^d$, while maintaining balance between columns.

To preserve the regular spacing between points, we apply the Cranley-Patterson rotation. This rotation involves shifting each row by a common random variable uniformly distributed in $u \sim [0,1)^d$. In effect, this corresponds to applying independent Cranley-Patterson rotations to each column of the variate matrix.

$$\begin{bmatrix} (a_1(1)+u_1) & \bmod 1 & (a_2(1)+u_2) & \bmod 1 & \dots & (a_d(1)+u_d) & \bmod 1 \\ (a_1(2)+u_1) & \bmod 1 & (a_2(2)+u_2) & \bmod 1 & \dots & (a_d(2)+u_d) & \bmod 1 \\ \vdots & & \vdots & & \ddots & \vdots & \\ (a_1(N)+u_1) & \bmod 1 & (a_2(N)+u_2) & \bmod 1 & \dots & (a_d(N)+u_d) & \bmod 1 \end{bmatrix}$$

Here, $a_j(i)$ represents the original CUD values in the matrix before randomization, and u_j is the randomization value for column j applied to each row. Each

element of the matrix is shifted by the corresponding randomization value, and then the result is taken modulo 1. As explained in 2.1.2, this randomization ensures that each point stays within the unit cube $[0,1)^d$ and preserves the marginal uniformity while maintaining the balance across columns.

Theorem 3.2 Cranley-Patterson rotations on CUD sequence

Given an array-CUD collection of generator sequences, defined by repeated values with regular skips and thinned to primes not congruent to 1 mod y_0, define the random array by applying d independent Cranley-Patterson rotations, where the n-th value undergoes the j-th rotation if $n \equiv j \mod d$. Then, the resulting array is WCUD.

This theorem, which is proven in [9], guarantees that by applying d independent Cranley-Patterson rotations to a CUD sequence, the resulting array maintains the CUD low-discrepancy properties array-wise.

The theorem applies to the CUD sequences generated by the methods described earlier in this section to ensure the preservation of the uniform distribution across the space. These generators, when processed with the Cranley-Patterson rotation, maintain the low-discrepancy property necessary for effective MCQMC algorithms, such as Metropolis-Hastings.

3.2 Implementation

In this section we aim at implementing a version of QMC Metropolis-Hastings inspired by the work of [8] and [2]. Then, we compare it on a toy example to the classic MCMC M-H. The goal is to estimate the mean of a **multivariate standard gaussian** with two different proposal approach:

1. A dependent proposal: $y_{i+1} = \Phi_i(X_i, u_{di+1}, \dots, u_{di+d})$ where the candidate depends on the current state of the Markov Chain and d *iid* uniform random variables in classic M-H (d points of a CUD sequence in the QMC version). We combine the current state with techniques such as inversion performed on the u_i.

2. A independent proposal: $y_{i+1} = \Phi_i(u_{di+1}, \dots, u_{di+d})$ where the candidate depends only on d *iid* uniform random variables in classic M-H (d points of a CUD sequence in the QMC version). Techniques such as inversion would be performed on the u_i.

We finally have the two following architectures to compare (we only present here the dependent versions):

Algorithm 1 Metropolis-Hastings IID uniforms dependent proposal

Input: Initial starting point x_0, Number of proposals $n_{samples}$, Target pdf π, Dimension d, Proposal std σ.

Output: Generated samples $\{x_n\}$ of size $n_{samples}$.

for *each MCMC iteration* $i = 1, 2, \ldots, n_{samples}$ **do**

- Sample d iid uniforms $u_1, \ldots, u_d$
- Generate a candidate $y_{i+1} = \Phi_i(x_i, u_1, \ldots, u_d)$ using the dependent proposal described before with σ
- Sample a new uniform U and update the chain if $U < \min\left(1, \frac{\pi(y_{i+1})}{\pi(x_i)}\right)$

return *samples*

Algorithm 2 Metropolis-Hastings CUD uniforms dependent proposal

Input: Initial starting point x_0, Number of proposals $n_{samples}$, Target pdf π, Dimension d, Proposal std σ.

Output: Generated samples $\{x_n\}$ of size $n_{samples}$.

Generate a CUD sequence of size $n_{samples} * (d+1)$ $u_1, \ldots, u_{n_{samples}*(d+1)}$ and reshape in the variate matrix of shape $= (n_{samples}, d+1)$ as explained in 3.1.

Apply the same randomization to each row of the variate matrix (Cranley-Patterson rotation) ψ.

for *each MCMC iteration* $i = 1, 2, \ldots, n_{samples}$ **do**

- Generate a candidate $y_{i+1} = \Phi_i(x_i, \psi_1(u_{di+1}), \ldots, \psi_d(u_{di+d}))$ using the dependent proposal described before with σ
- Update the chain if $u_{di+d+1} < \min\left(1, \frac{\pi(y_{i+1})}{\pi(x_i)}\right)$

return *samples*

For the dependent proposal we consider a random walk sampler i.e $y_{i+1} \sim \mathcal{N}(x_i, 2.4^2 * Id)$ for both methods. For the independent proposal we also consider a random walk sampler i.e $y_{i+1} \sim \mathcal{N}(0_d, 2.4^2 * Id)$ for both methods ($\sigma = 2.4$).

Our code is robust to all dimension but after learning about the properties of CUD sequences we may select wisely the parameter d and $n_{samples}$ for implementation.

The results presented here were obtain in dimension 2 and 21845 proposals ($n_{samples}$), running each method 100 times ($n_{runs} = 100$). We chose this values because the generator used only produces CUD sequences of size $2^m - 1$ with $10 \leq m \leq 32$ (see [6]). In these settings we thus have $2^m - 1 = n_{samples} * (d+1)$ with $m = 16$. This implies that we visit the entire CUD sequence to generate the $n_{samples}$ which is important because of nature of such CUD sequence (see 3.1). One can notice that we only need to generate the $2^m - 1$ CUD points once because we add variability later using a different randomization in each run.

As results we expose the average pool estimate over the 100 runs and the MSE of the 100 estimates against the true value ($\mu_{true} = [0, 0]$) and MSE = $\frac{1}{n_{\text{runs}}} \sum_{i=1}^{n_{\text{runs}}} \|\hat{\mu}_i - \mu_{\text{true}}\|^2$) in the following table.

Table 2 summarizes the results of the comparison (see *Appendix D* for code):

Method	Proposal Type	Pool Estimate	MSE
MC	Independente	[2.80e-03, -3.37e-04]	4.92e-04
QMC	Independente	[4.07e-04, 1.40e-03]	4.11e-04
MC	Dependente	[6.34e-04, 3.22e-03]	1.03e-03
QMC	Dependente	[-3.25e-03, -3.98e-03]	6.34e-04

Table 2: Comparison of MCMC and Q-MCMC.

In each case the pool estimation is close to μ_{true}. For the independente proposal type, the use of CUD points in M-H does not seem to have a significant MSE reduction effect. For the dependente proposal however, we can notice a small improvement for the MSE metric.

3.3 Conclusion

After exploring the theory of Q-MCMC and implementing an algorithm using this method, we have drawn several conclusions and identified some remaining open questions. One of the primary limitations of using CUD sequences in MCMC lies in the constraints imposed by the CUD generators, which provide sequences of "imposed sizes", and its non-trivial implementation.

Our results show that while the Q-MCMC implementation works as expected, the observed improvements in terms of MSE metrics are modest. This is consistent with the conclusions of [8], where the authors note that in applications satisfying regularity conditions, the benefits of CUD-driven MCMC are evident, though in some cases the advantages may not always be substantial. One possible explanation for the modest improvements in our results could be the use of the same CUD generator as in prior work, suggesting that further exploration of alternative or optimized CUD constructions may yield better outcomes.

Finally, various questions remain open and could be addressed empirically. For instance, how can we design an optimal randomization scheme that maximizes the benefits of CUD sequences in Q-MCMC?

References

[1] J. DICK ART B. OWEN, S. CHEN. Consistency of markov chain quasi-monte carlo on continuous state spaces. *Stanford University, University of*

New South Wales, 2011.

[2] Seth D. Tribble Art B. Owen. A quasi-monte carlo metropolis algorithm. *Department of Statistics, Stanford University*, April 2005.

[3] Christian Robert George Casella. Monte carlo statistical methods. 1999.

[4] Russel E. Caflisch Bradley Moskowitz. Modified monte carlo methods using quasi-random sequences. 1995.

[5] Art B. Owen. Practical quasi-monte carlo integration. *Department of Statistics, Stanford University*, January 2023 2023.

[6] S. Chen M. Matsumoto T. Nishimura A. B. Owen. New inputs and methods for markov chain quasi-monte carlo. *Monte Carlo and quasi-Monte Carlo methods*, 2010.

[7] Walter R Gilks Sylvia Richardson David J Spiegelhalter. Introducing markov chain monte carlo. 1996.

[8] Ben Calderhead Tobias Schwedes. Quasi markov chain monte carlo methods. October 2018.

[9] Seth D. Tribble. Markov chain monte carlo algorithms using completely uniformly distributed driving sequences. 2007.

A Annexe A : Python code for generating a CUD sequence by Tobias Schwedes

```
"""
Created on Mon May 28 19:22:20 2018

@author: Tobias Schwedes
"""

def chen_construction(m):

    """
    Constructs a 1-dimensional CUD-sequence of length 2^m-1 with 10 <= m <= 32
    according to the construction introduced in

        "New inputs and methods for Markov chain quasi-Monte Carlo"

    by

        Chen, Matsumoto, Nishimura and Owen (2012)

    and coefficient lists of primitive polynomials provided in

        "Primitive polynomials over finite fields"

    by

        Hansen and Mullen (1992)

    Input:
    m - length parameter

    Output:
    cuds - CUD sequence, also saved as .npy file
    """

    # Make sure valid CUD length parameter is chosen
    if m<10 or m>32:
        raise Exception('Oops! That was no valid CUD length parameter. '
                        'Choose integer among: 10 <= m <= 32.')

    # LFSR parameter tuples [m,s=s(m)] such that 2^m-1 and f^s(m) are coprime
    PrimList = [[10,115], [11,291], [12,171], [13,267], [14,332], [15,388],
            [16,283], [17,514], [18,698], [19,706], [20,1304], [21,920],
```

```
            [22,1336], [23,1236], [24,1511], [25,1445], [26,1906],
            [27,1875], [28,2573], [29,2633], [30,2423], [31,3573],
            ↪  [32,3632]]
    PrimArray = np.array(PrimList)

    # Get parameter s(m) from tuple array
    n = np.where(PrimArray==m)[0][0]
    s = PrimArray[n,1]

    # Define fractions of form 2^-i for binary representation
    bins = 2.**(-np.arange(1,m+1))

    # Set up array of CUD numbers
    cuds = np.zeros(2**m-1)
    print("Constructing a CUD sequence of length =", 2**m-1)

    # Starting coefficients
    s0 = np.ones(m)
    so = s0

    # Differentiate cases of constructions depending on m and s(m)
    if m==10:
        for i in range(2**m-1):
            for j in range(s):
                so = np.append(so[1:], (so[0]+so[3])%2)
            cuds[i] = np.sum(so*bins)
    elif m==11:
        for i in range(2**m-1):
            for j in range(s):
                so = np.append(so[1:], (so[0]+so[2])%2)
            cuds[i] = np.sum(so*bins)
    # Remaining cases omitted for brevity
    # Save CUD sequence as .npy
    #np.save('CudsChen_{}'.format(m), cuds)

    return cuds
```

B Annexe B : Python code for comparing plots of QMC sequence and IID sequence

```
def van_der_corput(n, base=2):
    """
    Generate the n-th Van der Corput number in the given base.
    """
    vdc = 0
    denom = 1
    while n > 0:
```

```
        n, remainder = divmod(n, base)
        denom *= base
        vdc += remainder / denom
    return vdc

def halton_sequence(dim, n_points):
    """
    Generate Halton sequence points.

    Parameters:
    - dim: The dimensionality of the Halton sequence.
    - n_points: The number of points to generate.

    Returns:
    - A (n_points x dim) array of Halton sequence points.
    """
    # Use the first `dim` prime numbers as bases
    primes = [prime(i) for i in range(1, dim + 1)]
    sequence = np.empty((n_points, dim))

    for d in range(dim):
        base = primes[d]
        sequence[:, d] = [van_der_corput(i, base) for i in
        ↪  range(n_points)]

    return sequence

    # Parameters for the plots
n_points_1 = 72       # Number of Halton points for the first plot
n_points_2 = 864      # Number of Halton points for the second and
↪  third plots
dim = 2               # Dimension of the Halton sequence

# Generate Halton points
halton_points_1 = halton_sequence(dim, n_points_1)
halton_points_2 = halton_sequence(dim, n_points_2)

# Generate random points for comparison
random_points = np.random.rand(n_points_2, dim)

# Create the plots
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# First panel: 72 Halton points
axes[0].scatter(halton_points_1[:, 0], halton_points_1[:, 1],
↪  c='black', s=40)
axes[0].set_title("72 Halton points", fontsize=16)
axes[0].set_xlim(0, 1)
axes[0].set_ylim(0, 1)
axes[0].set_aspect('equal')
```

```python

# Add grid lines for reference
for i in range(1, 9):  # 8 columns
    axes[0].axvline(i/8, color='gray', linestyle='--', linewidth=0.5)
for j in range(1, 9):  # 9 rows
    axes[0].axhline(j/9, color='gray', linestyle='--', linewidth=0.5)

# Second panel: 864 Halton points
axes[1].scatter(halton_points_2[:, 0], halton_points_2[:, 1],
↪  c='black', s=40)
axes[1].set_title("864 Halton points", fontsize=16)
axes[1].set_xlim(0, 1)
axes[1].set_ylim(0, 1)
axes[1].set_aspect('equal')

# Third panel: 864 random points
axes[2].scatter(random_points[:, 0], random_points[:, 1], c='black',
↪  s=40)
axes[2].set_title("864 random points", fontsize=16)
axes[2].set_xlim(0, 1)
axes[2].set_ylim(0, 1)
axes[2].set_aspect('equal')

# Final adjustments and display
for ax in axes:
    ax.set_xticks([])
    ax.set_yticks([])

plt.tight_layout()
plt.savefig("plot.pdf", format="pdf", bbox_inches="tight")
plt.show()

```

C Annexe C : Python code for comparing RQMC and MC method on toy example

```python
import numpy as np
import pandas as pd
from scipy.stats.qmc import Halton, Sobol

# Define the function to integrate
def f(x):
    return np.prod(np.cos(x), axis=1)

# Exact value of the integral
def exact_integral(d):
    return (np.sin(1) - np.sin(0)) ** d
```

```
12
13  # Monte Carlo estimation
14  def monte_carlo(f, n_samples, d):
15      x = np.random.rand(n_samples, d)
16      return np.mean(f(x))
17
18  # Randomized Quasi-Monte Carlo estimation (Halton or Sobol)
19  def rqmc(f, n_samples, d, method="halton"):
20      if method == "halton":
21          sampler = Halton(d=d, scramble=True)
22      elif method == "sobol":
23          sampler = Sobol(d=d, scramble=True)
24      x = sampler.random(n_samples)
25      return np.mean(f(x))
26
27  # Parameters
28  n_samples = 2**14  # Number of points per estimation
29  n_runs = 200       # Number of estimations per method
30  dimensions = [2, 24]  # Different dimensions to compare
31
32  # Collect results for all dimensions
33  results = []
34
35  for dimension in dimensions:
36      exact_value = exact_integral(dimension)
37
38      # Store results
39      mc_results = []
40      rqmc_results_sobol = []
41      rqmc_results_halton = []
42
43      for _ in range(n_runs):
44          mc_results.append(monte_carlo(f, n_samples, dimension))
45          rqmc_results_sobol.append(rqmc(f, n_samples, dimension,
            ↪  method="sobol"))
46          rqmc_results_halton.append(rqmc(f, n_samples, dimension,
            ↪  method="halton"))
47
48      # Calculate mean and RMSE of estimates
49      mc_mean = np.mean(mc_results)
50      mc_variance = np.var(mc_results)
51      mc_RMSE = np.sqrt(np.mean((mc_results - exact_value)**2))
52
53      rqmc_sobol_mean = np.mean(rqmc_results_sobol)
54      rqmc_sobol_variance = np.var(rqmc_results_sobol)
55      rqmc_sobol_RMSE = np.sqrt(np.mean((rqmc_results_sobol -
        ↪  exact_value)**2))
56
57      rqmc_halton_mean = np.mean(rqmc_results_halton)
58      rqmc_halton_variance = np.var(rqmc_results_halton)
```

```
    rqmc_halton_RMSE = np.sqrt(np.mean((rqmc_results_halton - exact_value)**2))

    # Append results
    results.append(["Monte Carlo", dimension, mc_mean, mc_RMSE**2, exact_value])
    results.append(["RQMC Halton", dimension, rqmc_halton_mean, rqmc_sobol_RMSE**2, exact_value])
    results.append(["RQMC Sobol", dimension, rqmc_sobol_mean, rqmc_halton_RMSE**2, exact_value])

# Create DataFrame
results_table = pd.DataFrame(
    results,
    columns=["Method", "Dimension", "Pool Estimate", "MSE", "Exact Value"]
)

# Display the table
results_table

```

D Annexe D : Python code for comparing MCMC and Q-MCMC on a toy example

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm, multivariate_normal
import math
from scipy.stats import norm



# Target distribution: Multivariate Standard normal (PDF)
def target_distribution(x):

    mv_normal = multivariate_normal(mean=np.zeros(len(x)), cov=np.eye(len(x)))

    return mv_normal.pdf(x)


# Random walk using gaussian transformation of uniforms
def random_walk_proposal(x, u, proposal_std):
    """
    Generate a random walk proposal using a Gaussian distribution, where the random
```

```
    numbers are generated from a uniform distribution and transformed
    ↪ to a normal distribution.

    Parameters:
    - x: Current state (scalar or array).
    - u: Uniform random variables in [0, 1], of the same shape as `x`.
    - proposal_std: Standard deviation of the Gaussian proposal.

    Returns:
    - Proposed state.
    """
    # Transform uniform variables u to standard normal variables using
    ↪ the probit function
    standard_normal = norm.ppf(u)  # Inverse CDF of the normal
    ↪ distribution

    # Scale the standard normal variables by the proposal_std and add
    ↪ to the current state
    proposed_state = x + proposal_std * standard_normal
    return proposed_state


def independent_proposal(u, proposal_std):
    """
    Generate a proposed state for Metropolis-Hastings using an
    ↪ independent proposal mechanism.

    Parameters:
    - u: Array of uniformly distributed random variables in the range
    ↪ [0, 1].
         These uniform variables are used to generate standard normal
         ↪ variables via the probit function.
    - proposal_std: Standard deviation of the proposal distribution.

    Returns:
    - proposed_state: The proposed state generated by scaling the
    ↪ standard normal variables
                      by the proposal standard deviation.

    Steps:
    1. Transform uniform random variables `u` into standard normal
    ↪ variables using the inverse CDF
       (probit function) of the standard normal distribution. This
       ↪ ensures the resulting variables
       have a normal distribution with mean 0 and variance 1.
    2. Scale these standard normal variables by the `proposal_std` to
    ↪ adjust the spread of the proposal distribution.
    3. Return the proposed state.
    """

```

```
    # Transform uniform variables u to standard normal variables using the probit function
    standard_normal = norm.ppf(u)  # Inverse CDF of the normal distribution

    # Scale the standard normal variables by the proposal_std
    proposed_state = proposal_std * standard_normal

    return proposed_state


def cranley_patterson_rotation(point):
    """
    Apply a Cranley-Patterson rotation to a given point in the unit hypercube [0, 1]^d.
    This randomization technique is used to maintain low-discrepancy properties while introducing randomness.

    Parameters:
    - point: A numpy array of shape (d,) representing a point in the d-dimensional unit hypercube [0, 1]^d.

    Returns:
    - rotated_point: A numpy array of shape (d,) representing the rotated point,
                     which also lies within the unit hypercube [0, 1]^d.

    Steps:
    1. Determine the dimension `d` of the input point.
    2. Generate a random vector in the unit hypercube [0, 1]^d.
       This vector acts as the rotation offset.
    3. Perform the Cranley-Patterson rotation by adding the random vector to the input point
       and wrapping the result modulo 1 to ensure the rotated point stays within the unit hypercube.
    4. Return the rotated point.
    """

    d = point.shape[0]  # Dimension of the point
    random_vector = np.random.rand(d)  # Generate random vector in [0, 1]^d

    # Apply the rotation and wrap values to remain within [0, 1]
    rotated_point = (point + random_vector) % 1
    return rotated_point


# Metropolis-Hastings algorithm with dependent proposal and IID uniforms
```

```
99  def metropolis_hastings_dependent_iid(target_pdf, dimension, n_samples,
    ↪  initial, proposal_std=0.1):
100     """
101     Implements the Metropolis-Hastings algorithm using dependent
        ↪  proposals and IID uniform random variables.
102
103     Parameters:
104     - target_pdf: Callable
105       The target probability density function to sample from.
106     - dimension: int
107       The dimensionality of the target distribution.
108     - n_samples: int
109       The number of samples to generate.
110     - initial: numpy array
111       The starting point for the Markov chain, with shape (dimension,).
112     - proposal_std: float, optional (default=0.1)
113       The standard deviation of the proposal distribution for the
          ↪  random walk.
114
115     Returns:
116     - samples: numpy array
117       An array of shape (n_samples + 1, dimension) containing the
          ↪  generated samples.
118
119     Algorithm Steps:
120     1. Initialize an array `samples` to store the generated points. Set
        ↪  the first sample to `initial`.
121     2. For each iteration (up to `n_samples`):
122         a. Generate a random candidate point using the
            ↪  `random_walk_proposal` function,
123            which uses IID uniform random variables for the proposal.
124         b. Compute the acceptance ratio, which is the ratio of the
            ↪  target PDF at the candidate point
125            to the target PDF at the current point, bounded by 1.
126         c. Accept the candidate with a probability equal to the
            ↪  acceptance ratio. If rejected,
127            retain the current point as the next sample.
128     3. Return the array of samples.
129
130     Notes:
131     - The proposal is a random walk, where the next state depends on
        ↪  the current state and IID uniform noise.
132     - The use of uniform random variables ensures reproducibility in a
        ↪  controlled random sampling setting.
133
134     Example:
135     >>> import numpy as np
136     >>> def target_pdf(x):
137     >>>     return np.exp(-0.5 * np.sum(x**2, axis=-1))  # Standard
        ↪  normal PDF
```

```
    >>> samples = metropolis_hastings_dependent_iid(target_pdf,
    ↪  dimension=2, n_samples=1000, initial=np.zeros(2))
    >>> print(samples.shape)  # Output: (1001, 2)
    """

    # Initialize the array to store the samples
    samples = np.zeros((n_samples + 1, dimension))
    samples[0] = initial  # Set the initial sample

    # Perform the Metropolis-Hastings sampling
    for i in range(n_samples):
        # Generate IID uniform random variables
        iid_uniforms = np.random.rand(dimension)

        # Generate a candidate using the random walk proposal
        candidate = random_walk_proposal(samples[i], iid_uniforms,
        ↪  proposal_std)

        # Compute the acceptance ratio
        acceptance_ratio = min(1, target_pdf(candidate) /
        ↪  target_pdf(samples[i]))

        # Accept or reject the candidate
        if np.random.rand() < acceptance_ratio:
            samples[i+1] = candidate  # Accept the candidate
        else:
            samples[i+1] = samples[i]  # Reject and keep the current
            ↪  sample

    return np.array(samples)


# Metropolis-Hastings algorithm with dependent proposal and IID
↪  uniforms
def metropolis_hastings_independent_iid(target_pdf, dimension,
↪  n_samples, initial, proposal_std=0.1):
    """
    Implements the Metropolis-Hastings algorithm using independent
    ↪  proposals and IID uniform random variables.

    Parameters:
    - target_pdf: Callable
      The target probability density function to sample from.
    - dimension: int
      The dimensionality of the target distribution.
    - n_samples: int
      The number of samples to generate.
    - initial: numpy array
      The starting point for the Markov chain, with shape (dimension,).
    - proposal_std: float, optional (default=0.1)
```

```
    The standard deviation of the proposal distribution.

Returns:
- samples: numpy array
  An array of shape (n_samples + 1, dimension) containing the generated samples.

Algorithm Steps:
1. Initialize an array `samples` to store the generated points. Set the first sample to `initial`.
2. For each iteration (up to `n_samples`):
    a. Generate a random candidate point using the `independent_proposal` function,
       which uses IID uniform random variables to generate proposals independently of the current state.
    b. Compute the acceptance ratio, which is the ratio of the target PDF at the candidate point
       to the target PDF at the current point, bounded by 1.
    c. Accept the candidate with a probability equal to the acceptance ratio. If rejected,
       retain the current point as the next sample.
3. Return the array of samples.

Notes:
- The proposal is independent of the current state.
- Uniform random variables are transformed into proposal points using the `independent_proposal` function.

Example:
>>> import numpy as np
>>> def target_pdf(x):
>>>     return np.exp(-0.5 * np.sum(x**2, axis=-1))  # Standard normal PDF
>>> samples = metropolis_hastings_independent_iid(target_pdf, dimension=2, n_samples=1000, initial=np.zeros(2))
>>> print(samples.shape)  # Output: (1001, 2)
"""

# Initialize the array to store the samples
samples = np.zeros((n_samples + 1, dimension))
samples[0] = initial  # Set the initial sample

# Perform the Metropolis-Hastings sampling
for i in range(n_samples):
    # Generate IID uniform random variables
    iid_uniforms = np.random.rand(dimension)

    # Generate a candidate independently of the current state
    candidate = independent_proposal(iid_uniforms, proposal_std)

```

```
222             # Compute the acceptance ratio
223             acceptance_ratio = min(1, target_pdf(candidate) /
                ↪  target_pdf(samples[i]))
224
225             # Accept or reject the candidate
226             if np.random.rand() < acceptance_ratio:
227                 samples[i+1] = candidate  # Accept the candidate
228             else:
229                 samples[i+1] = samples[i]  # Reject and keep the current
                    ↪  sample
230
231         return np.array(samples)
232
233
234     def find_optimal_m(n_samples, dim):
235         """
236         Find the optimal value of `m` for constructing a Chen CUD sequence.
237         The goal is to ensure the generated sequence contains at least the
            ↪  required number of points,
238         while minimizing excess points for efficiency.
239
240         Parameters:
241         - n_samples: int
242           The number of desired samples for the Monte Carlo simulation.
243         - dim: int
244           The dimensionality of the target distribution or the number of
              ↪  dimensions for each sample.
245
246         Returns:
247         - m: int
248           The smallest integer `m` such that the Chen CUD sequence has at
              ↪  least `n_samples * (dim + 1)` points.
249
250         Example:
251         >>> find_optimal_m(n_samples=1000, dim=3)
252         12  # Returns the smallest m such that 2^m - 1 >= 1000 * (3 + 1)
253
254         Notes:
255         - The extra factor `dim + 1` accounts for any additional uniform
            ↪  variables required
256           for acceptance or proposal mechanisms.
257         - This function minimizes the computational overhead of generating
            ↪  excess CUD points.
258         """
259
260         # Initial calculation of m
261         m = math.ceil(math.log2(n_samples * (dim + 1) + 1))
262
263         # Check if m - 1 is sufficient
264         if (2**(m-1) - 1) >= n_samples * (dim + 1):
```

```
265             m -= 1  # Decrease m if possible
266
267     return m
268
269
270 # Metropolis-Hastings algorithm with dependent proposal and CUD
    ↪ uniforms
271 def metropolis_hastings_dependent_CUD(target_pdf, dimension, n_samples,
    ↪ initial, m, CUD_points, proposal_std=0.1):
272     """
273     Implements the Metropolis-Hastings algorithm with dependent
        ↪ proposals using CUD (Completely Uniformly Distributed) points.
274
275     Parameters:
276     - target_pdf: Callable
277       The target probability density function to sample from.
278     - dimension: int
279       The dimensionality of the target distribution.
280     - n_samples: int
281       The number of samples to generate.
282     - initial: numpy array
283       The starting point for the Markov chain, with shape (dimension,).
284     - m: int
285       The parameter defining the length of the CUD sequence, where the
          ↪ total number of points is 2^m - 1.
286     - CUD_points: numpy array
287       A 1D array of pre-generated CUD points of length >= n_samples *
          ↪ (dimension + 1).
288     - proposal_std: float, optional (default=0.1)
289       The standard deviation of the proposal distribution.
290
291     Returns:
292     - samples: numpy array
293       An array of shape (n_samples + 1, dimension) containing the
          ↪ generated samples.
294
295     Algorithm:
296     1. Assert that `CUD_points` contains enough points to support the
        ↪ requested number of samples and dimensions.
297        We need `n_samples * (dimension + 1)` points to perform
            ↪ proposals and acceptances.
298     2. Reshape `CUD_points` into a 2D array of shape `(n_samples,
        ↪ dimension + 1)`, where:
299        - The first `dimension` elements are used for the proposal.
300        - The last element is used for acceptance.
301     3. Apply Cranley-Patterson rotation to each group of CUD points to
        ↪ introduce randomization while maintaining low-discrepancy
        ↪ properties.
302     4. For each sample:
303         a. Extract the proposal and acceptance CUD points.
```

```
        b. Generate a candidate using the `random_walk_proposal`
        ↪ function.
        c. Compute the acceptance ratio using the target PDF.
        d. Accept the candidate if the acceptance CUD value is less
        ↪ than the acceptance ratio; otherwise, keep the current
        ↪ sample.
    5. Return the array of generated samples.

    Notes:
    - Randomization ensures variability between runs while preserving
    ↪ the low-discrepancy nature of the CUD sequence.
    - The proposal is a dependent random walk, but the CUD points
    ↪ ensure stratified sampling for better convergence.

    Example:
    >>> import numpy as np
    >>> def target_pdf(x):
    >>>     return np.exp(-0.5 * np.sum(x**2, axis=-1))  # Standard
    ↪ normal PDF
    >>> CUD_points = chen_construction(m=12)
    >>> samples = metropolis_hastings_dependent_CUD(target_pdf, 2,
    ↪ 1000, np.zeros(2), 12, CUD_points, 0.1)
    >>> print(samples.shape)  # Output: (1001, 2)
    """

    # Initialize the array to store the samples
    samples = np.zeros((n_samples + 1, dimension))
    samples[0] = initial  # Set the initial sample

    # Ensure we have enough CUD points for the requested samples and
    ↪ dimensions
    assert len(CUD_points) >= n_samples * (dimension + 1), \
        "Not enough CUD points for the requested number of samples and
        ↪ dimensions."

    # Reshape CUD points to use for proposals and acceptances
    CUD_points = np.reshape(CUD_points[:n_samples * (dimension + 1)],
    ↪ (n_samples, dimension + 1))

    # Apply Cranley-Patterson rotation to each group of CUD points
    v = np.random.rand(dimension + 1)
    for j in range(n_samples):
        CUD_points[j, :] = (CUD_points[j, :] + v) % 1

    # Perform the Metropolis-Hastings sampling
    for i in range(n_samples):
        # Extract current CUD points for proposal and acceptance
        current_CUD_points = CUD_points[i, :]

        proposal_CUD = current_CUD_points[:dimension]  # Proposal part
```

```
344         acceptance_CUD = current_CUD_points[dimension]  # Acceptance
            ↪ part
345
346         # Generate a candidate using the random walk proposal
347         candidate = random_walk_proposal(samples[i], proposal_CUD,
            ↪ proposal_std)
348
349         # Compute the acceptance ratio
350         acceptance_ratio = min(1, target_pdf(candidate) /
            ↪ target_pdf(samples[i]))
351
352         # Accept or reject the candidate
353         samples[i + 1] = candidate if acceptance_CUD < acceptance_ratio
            ↪ else samples[i]
354
355     return np.array(samples)
356
357
358 # Metropolis-Hastings algorithm with dependent proposal and CUD
    ↪ uniforms
359 def metropolis_hastings_independent_CUD(target_pdf, dimension,
    ↪ n_samples, initial, m, CUD_points, proposal_std=0.1):
360     """
361     Implements the Metropolis-Hastings algorithm with independent
        ↪ proposals using CUD (Completely Uniformly Distributed) points.
362
363     Parameters:
364     - target_pdf: Callable
365       The target probability density function to sample from.
366     - dimension: int
367       The dimensionality of the target distribution.
368     - n_samples: int
369       The number of samples to generate.
370     - initial: numpy array
371       The starting point for the Markov chain, with shape (dimension,).
372     - m: int
373       The parameter defining the length of the CUD sequence, where the
          ↪ total number of points is 2^m - 1.
374     - CUD_points: numpy array
375       A 1D array of pre-generated CUD points of length >= n_samples *
          ↪ (dimension + 1).
376     - proposal_std: float, optional (default=0.1)
377       The standard deviation of the proposal distribution.
378
379     Returns:
380     - samples: numpy array
381       An array of shape (n_samples + 1, dimension) containing the
          ↪ generated samples.
382
383     Algorithm:
```

```
    1. Assert that `CUD_points` contains enough points to support the requested number of samples and dimensions.
       We need `n_samples * (dimension + 1)` points to perform proposals and acceptances.
    2. Reshape `CUD_points` into a 2D array of shape `(n_samples, dimension + 1)`, where:
       - The first `dimension` elements are used for the proposal.
       - The last element is used for acceptance.
    3. Apply Cranley-Patterson rotation to each group of CUD points to introduce randomization while maintaining low-discrepancy properties.
    4. For each sample:
        a. Extract the proposal and acceptance CUD points.
        b. Generate a candidate using the `independent_proposal` function, which creates proposals independent of the current state.
        c. Compute the acceptance ratio using the target PDF.
        d. Accept the candidate if the acceptance CUD value is less than the acceptance ratio; otherwise, keep the current sample.
    5. Return the array of generated samples.

    Notes:
    - Randomization ensures variability between runs while preserving the low-discrepancy nature of the CUD sequence.
    - The proposal is independent, meaning the candidate does not depend on the current state.

    Example:
    >>> import numpy as np
    >>> def target_pdf(x):
    >>>     return np.exp(-0.5 * np.sum(x**2, axis=-1))  # Standard normal PDF
    >>> CUD_points = chen_construction(m=12)
    >>> samples = metropolis_hastings_independent_CUD(target_pdf, 2, 1000, np.zeros(2), 12, CUD_points, 0.1)
    >>> print(samples.shape)  # Output: (1001, 2)
    """

    # Initialize the array to store the samples
    samples = np.zeros((n_samples + 1, dimension))
    samples[0] = initial  # Set the initial sample

    # Ensure we have enough CUD points for the requested samples and dimensions
    assert len(CUD_points) >= n_samples * (dimension + 1), \
        "Not enough CUD points for the requested number of samples and dimensions."

    # Reshape CUD points to use for proposals and acceptances
```

```
    CUD_points = np.reshape(CUD_points[:n_samples * (dimension + 1)],
    ↪  (n_samples, dimension + 1))

    # # Apply Cranley-Patterson rotation to each group of CUD points
    v = np.random.rand(dimension + 1)
    for j in range(n_samples):
        CUD_points[j, :] = (CUD_points[j, :] + v) % 1

    # Perform the Metropolis-Hastings sampling
    for i in range(n_samples):
        # Extract current CUD points for proposal and acceptance
        current_CUD_points = CUD_points[i, :]

        proposal_CUD = current_CUD_points[:dimension]  # Proposal part
        acceptance_CUD = current_CUD_points[dimension]  # Acceptance
        ↪  part

        # Generate a candidate using the independent proposal mechanism
        candidate = independent_proposal(proposal_CUD, proposal_std)

        # Compute the acceptance ratio
        acceptance_ratio = min(1, target_pdf(candidate) /
        ↪  target_pdf(samples[i]))

        # Accept or reject the candidate
        samples[i + 1] = candidate if acceptance_CUD < acceptance_ratio
        ↪  else samples[i]

    return np.array(samples)

# Dimension of the target distribution
dim = 2

# List of sample sizes for which the RMSE will be computed
n_samples_list = np.linspace(21845, 21845, num=1, dtype=int).tolist()

# Initial state and parameters for the proposal distribution
initial_value = np.zeros(dim)
proposal_std = 2.4

# Burn-in period: discard the first 10% of samples
burn_in = int(0.1 * n_samples)

# Containers for RMSE results for different methods
iid_MH_results_RMSE_dependent = []      # RMSE for IID-dependent
↪  proposals
CUD_MH_results_RMSE_dependent = []      # RMSE for CUD-dependent
↪  proposals
iid_MH_results_RMSE_independent = []    # RMSE for IID-independent
↪  proposals
```

```
CUD_MH_results_RMSE_independent = []    # RMSE for CUD-independent proposals

iid_MH_results_POOL_ESTIMATE_dependent = []      # POOL AVERAGE ESTIMATE for IID-dependent proposals
CUD_MH_results_POOL_ESTIMATE_dependent = []      # POOL AVERAGE ESTIMATE for CUD-dependent proposals
iid_MH_results_POOL_ESTIMATE_independent = []    # POOL AVERAGE ESTIMATE for IID-independent proposals
CUD_MH_results_POOL_ESTIMATE_independent = []    # POOL AVERAGE ESTIMATE for CUD-independent proposals

# Number of independent runs for RMSE computation
n_runs = 100

# True mean of the target distribution
true_mean = np.zeros(dim)

# Loop over different sample sizes
for n_samples in n_samples_list:

    # Find optimal `m` for the required number of CUD points
    m = find_optimal_m(n_samples, dim)

    # Generate CUD points using Chen's construction
    CUD_points = chen_construction(m)

    # Containers for results of individual runs
    iid_MH_results_dependent = []
    CUD_MH_results_dependent = []
    iid_MH_results_independent = []
    CUD_MH_results_independent = []

    # Perform `n_runs` independent Metropolis-Hastings simulations
    for i in range(n_runs):
        # Dependent proposal with IID uniforms
        samples_iid_dependent = metropolis_hastings_dependent_iid(target_distribution, dim, n_samples, initial_value, proposal_std)
        # Dependent proposal with CUD uniforms
        samples_CUD_dependent = metropolis_hastings_dependent_CUD(target_distribution, dim, n_samples, initial_value, m, CUD_points, proposal_std)

        # Compute the mean of the retained samples for dependent proposals
        mean_estimate_iid_dependent = np.mean(samples_iid_dependent[burn_in:], axis=0)
        mean_estimate_CUD_dependent = np.mean(samples_CUD_dependent[burn_in:], axis=0)
```

```
500
501         # Store the results
502         iid_MH_results_dependent.append(mean_estimate_iid_dependent)
503         CUD_MH_results_dependent.append(mean_estimate_CUD_dependent)
504
505         # Independent proposal with IID uniforms
506         samples_iid_independent =
            ↪  metropolis_hastings_independent_iid(target_distribution,
            ↪  dim, n_samples, initial_value, proposal_std)
507         # Independent proposal with CUD uniforms
508         samples_CUD_independent =
            ↪  metropolis_hastings_independent_CUD(target_distribution,
            ↪  dim, n_samples, initial_value, m, CUD_points, proposal_std)
509
510         # Compute the mean of the retained samples for independent
            ↪  proposals
511         mean_estimate_iid_independent =
            ↪  np.mean(samples_iid_independent[burn_in:], axis=0)
512         mean_estimate_CUD_independent =
            ↪  np.mean(samples_CUD_independent[burn_in:], axis=0)
513
514         # Store the results
515
            ↪  iid_MH_results_independent.append(mean_estimate_iid_independent)
516
            ↪  CUD_MH_results_independent.append(mean_estimate_CUD_independent)
517
518         print(i)  # Track progress of the simulation runs
519
520     # Convert the collected results into numpy arrays
521     iid_MH_results_dependent = np.array(iid_MH_results_dependent)
522     CUD_MH_results_dependent = np.array(CUD_MH_results_dependent)
523     iid_MH_results_independent = np.array(iid_MH_results_independent)
524     CUD_MH_results_independent = np.array(CUD_MH_results_independent)
525
526     # Compute the pool estimate over the n_runs
527
        ↪  iid_MH_results_POOL_ESTIMATE_dependent.append(np.mean(iid_MH_results_dependent,
        ↪  axis=0))
528
        ↪  CUD_MH_results_POOL_ESTIMATE_dependent.append(np.mean(CUD_MH_results_dependent,
        ↪  axis=0))
529
        ↪  iid_MH_results_POOL_ESTIMATE_independent.append(np.mean(iid_MH_results_independent,
        ↪  axis=0))
530
        ↪  CUD_MH_results_POOL_ESTIMATE_independent.append(np.mean(CUD_MH_results_independent,
        ↪  axis=0))
531
532
```

```
    # Compute RMSE for dependent proposals
    iid_rmse_dependent = np.sqrt(np.mean(np.sum((iid_MH_results_dependent - true_mean) ** 2, axis=1)))
    cud_rmse_dependent = np.sqrt(np.mean(np.sum((CUD_MH_results_dependent - true_mean) ** 2, axis=1)))
    iid_MH_results_RMSE_dependent.append(iid_rmse_dependent)
    CUD_MH_results_RMSE_dependent.append(cud_rmse_dependent)

    # Compute RMSE for independent proposals
    iid_rmse_independent = np.sqrt(np.mean(np.sum((iid_MH_results_independent - true_mean) ** 2, axis=1)))
    cud_rmse_independent = np.sqrt(np.mean(np.sum((CUD_MH_results_independent - true_mean) ** 2, axis=1)))
    iid_MH_results_RMSE_independent.append(iid_rmse_independent)
    CUD_MH_results_RMSE_independent.append(cud_rmse_independent)

import pandas as pd

# Create a dictionary to store the results
results = {
    "Method": [],
    "Proposal Type": [],
    "Pool Estimate": [],
    "MSE": []
}

# Function to format vectors (e.g., Pool Estimate) for readability
def format_vector(vector):
    return "[" + ", ".join(f"{x:.2e}" for x in vector) + "]"

# Populate results for the Independent sampler
results["Method"].append("MC")
results["Proposal Type"].append("Independente")
results["Pool Estimate"].append(format_vector(iid_MH_results_POOL_ESTIMATE_independent[0]))
results["MSE"].append(f"{iid_rmse_independent**2:.2e}")

results["Method"].append("QMC")
results["Proposal Type"].append("Independente")
results["Pool Estimate"].append(format_vector(CUD_MH_results_POOL_ESTIMATE_independent[0]))
results["MSE"].append(f"{cud_rmse_independent**2:.2e}")

# Populate results for the Random Walk sampler
results["Method"].append("MC")
results["Proposal Type"].append("Dependente")
```

```
results["Pool
↪  Estimate"].append(format_vector(iid_MH_results_POOL_ESTIMATE_dependent[0]))
results["MSE"].append(f"{iid_rmse_dependent**2:.2e}")

results["Method"].append("QMC")
results["Proposal Type"].append("Dependente")
results["Pool
↪  Estimate"].append(format_vector(CUD_MH_results_POOL_ESTIMATE_dependent[0]))
results["MSE"].append(f"{cud_rmse_dependent**2:.2e}")

# Convert the dictionary to a DataFrame
results_df = pd.DataFrame(results)

results_df

```